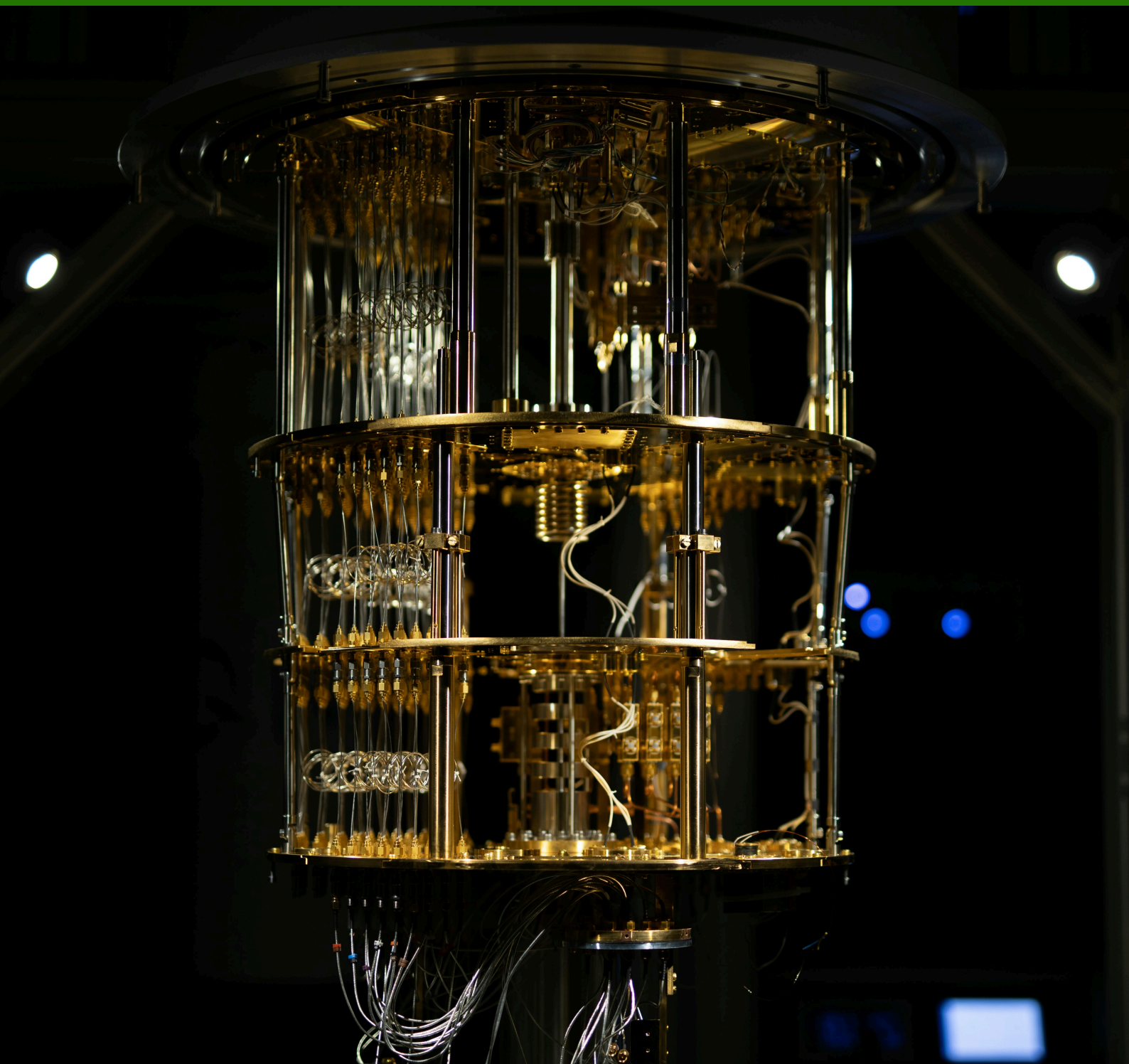
Dipartimento di Informatica
Università degli Studi di Verona
Verona (VR), Italia

CrossQuil
Realizzazione di una libreria per lo sviluppo di programmi quantistici in più linguaggi

Tesi per il Corso di Laurea in Informatica

Autore: Sacchetto Riccardo
Relatrici: Prof.ssa Daffara Claudia Prof.ssa Di Pierro Alessandra



Abstract

Visto l'attuale stato dell'arte delle librerie per lo sviluppo di programmi quantistici e la loro tendenza a concentrarsi sul linguaggio di programmazione Python, lo scopo di questo lavoro è la realizzazione di un framework che consenta di scrivere, simulare e mandare in esecuzione su hardware reale circuiti quantistici basati sull'ecosistema Quil in un qualunque linguaggio di programmazione.

Tale obiettivo sarà raggiunto mediante un'architettura incentrata su una libreria a basso livello realizzata in Rust e una interfaccia API basata sulla FFI di C.

DOI: 10.5281/zenodo.17316099

Indice

Introduzione 3
- I programmi quantistici 3
- Quil 3
- Architettura generale 4

Astrazioni 5
- Un semplice programma Quil 5
- Composizione di un programma quantistico 5
- Interazione con compilatore e QVM 7

Integrazione con Azure Quantum 9
- Funzionamento di Azure Quantum 9
- Organizzazione del client 9

Creazione della FFI 11
- Scelte architetturali 11
- Esposizione di una struttura dati 11
- Considerazioni di sicurezza 13

Integrazione di esempio: Java (+ MATLAB) 14
- Lifecycle di una classe CrossQuil in Java 14
- Importazione di array e matrici 15
- Passaggio a MATLAB 16

La CrossQVM 17
- L'architettura della CrossQVM 17
- Rappresentazione dello stato dei qubit 17
- Lifting di un gate 18
- Operazioni su qubit non adiacenti 20
- Misurazione di un qubit 20
- Esecuzione di un programma 21

Conclusione e sviluppi futuri 23

Bibliografia 24

Introduzione

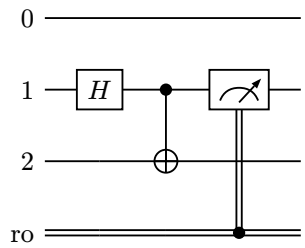
Negli ultimi anni, l'informatica quantistica ha visto una rapida crescita sia dal punto di vista teorico che applicativo, grazie allo sviluppo di nuovi algoritmi e all'accessibilità di hardware quantistico. Tuttavia, la maggior parte delle librerie e dei framework per la programmazione quantistica si concentra prevalentemente sull'ecosistema Python, limitando così la possibilità di integrazione con altri linguaggi di programmazione e ambienti di sviluppo.

Questo progetto nasce inizialmente dalla proposta di realizzare il primo framework quantistico in MATLAB, con l'obiettivo di consentire anche a chi si trova più a suo agio con questo liguaggio di scrivere e integrare con facilità algoritmi quantistici. Nel corso dello sviluppo, tuttavia, il progetto si è evoluto per abbracciare una visione più ampia: creare un framework versatile che permetta di scrivere, simulare ed eseguire circuiti quantistici basati sull'ecosistema Quil in qualsiasi linguaggio di programmazione.

I programmi quantistici

Un programma quantistico è, nella sua essenza, l'implementazione di un circuito quantistico: una sequenza ordinata di operazioni (gate) che vengono applicate a un insieme di qubit, le unità fondamentali dell'informazione quantistica. Questi circuiti sono progettati per sfruttare le proprietà uniche della meccanica quantistica, come la sovrapposizione e l'entanglement, al fine di risolvere problemi che risultano intrattabili per i computer classici.

Ad esempio, quello che segue è un circuito quantistico che prepara uno stato di Bell tra due qubit, misura il secondo qubit e memorizza il risultato in un registro classico:



La descrizione di un programma quantistico avviene tipicamente attraverso un linguaggio formale che consente di specificare le operazioni da eseguire su ogni qubit, le misurazioni da effettuare e le eventuali interazioni con la memoria classica. L'esecuzione di un circuito quantistico può avvenire sia su simulatori software che su hardware quantistico reale, permettendo di studiare e testare algoritmi quantistici in diversi contesti applicativi.

Quil

Quil (Quantum Instruction Language) è un Quantum Instruction Set Architecture (QISA) sviluppato da Robert Smith, Michael Curtis e William Zeng in *A Practical Quantum Instruction Set Architecture*[1] per Rigetti Computing, progettato specificamente per la descrizione e l'esecuzione di circuiti quantistici. Primo QISA a introdurre un modello di memoria ibridio quantistico-classico, consente di rappresentare sia operazioni quantistiche che classiche.

Uno degli aspetti più rilevanti dell'ecosistema Quil ai fini di questo progetto è la disponibilità di API implementate in Rust, che, oltre ad astrarre le varie istruzioni supportate da Quil, permettono di interagire con facilità con il compilatore, la Quantum Virtual Machine (QVM) e l'hardware reale sviluppati in-house da Rigetti. Grazie a queste API, infatti, è stato possibile realizzare un framework che non solo sfrutta l'efficienza e la sicurezza offerte da Rust e che può essere facilmente integrato con

[1] Robert S. Smith and Michael J. Curtis and William J. Zeng, «A Practical Quantum Instruction Set Architecture», 17 febbraio 2017, arXiv, https://arxiv.org/abs/1608.03355.

altri linguaggi tramite una Foreign Function Interface (FFI), ma che è in grado di mettere a disposizione dell'utente finale strumenti e computer quantistici di uso reale.

Architettura generale

Il cuore del framework si trova all'interno della libreria Rust, `libcrossquil`. Questa, suddivisa a sua volta nelle componenti `base` e `c_ffi`, definisce sia tutte le funzioni e gli struct necessari per la manipolazione e l'esecuzione di circuiti quantistici in Quil, sia l'interfaccia C che permette di esporre tali funzionalità a linguaggi di programmazione esterni.

All'interno di `base` sono presenti, essenzialmente, l'astrazione del linguaggio Quil e le implementazioni di Quantum Abstract Machine (QAM) supportate da CrossQuil. Qui si trova tutto il codice che può essere condiviso tra le varie piattaforme «target» del framework e che, pertanto, non dovrà essere implementato da capo per ciascuna di esse.

All'interno di `c_ffi` sono invece raccolte tutte quelle funzioni che espongono il contenuto di `base` secondo la logica di C, che in informatica è considerato essenzialmente la lingua franca dei linguaggi di programmazione. Operando esclusivamente con puntatori allo heap, queste sono in grado di fornire e manipolare gli oggetti Rust per conto di linguaggi esterni, i quali potranno trattarli semplicemente come «black box».

Astrazioni

Il primo passo logico per realizzare il progetto che stiamo descrivendo è stato, ovviamente, la realizzazione dell'astrazione attorno a Quil e alle possibili implementazioni della QAM, elemento fondamentale per mettere a disposizione degli utenti l'interfaccia finale con cui interagire.

Partiamo dunque col descrivere come questa è stata realizzata e come si comporta, iniziando dallo studio di un semplice programma Quil e di come questo può essere trattato dall'astrazione che vogliamo costruire.

Un semplice programma Quil

Ai fini della nostra trattazione analizzeremo il programma Quil che implementa il circuito visto nell'introduzione:

```
DECLARE ro BIT[1]
I 0
H 1
CNOT 1 2
MEASURE 1 ro[0]
```

Questo programma, pensato per essere eseguito su un Quantum Computer a tre qubit, risulta nell'esecuzione delle seguenti operazioni, una per ogni riga di codice, partendo dallo stato $|\psi\rangle = |000\rangle$:

1. Dichiarazione di un registro classico `ro` di un bit, che verrà usato per memorizzare il risultato della misura del qubit 1;
2. Applicazione dell'operazione di identità (I) sul qubit 0 (necessario per assicurarsi che la QVM operi effettivamente su 3 qubit);
3. Applicazione della porta di Hadamard (H) sul qubit 1, mettendolo in sovrapposizione ($|\psi\rangle = \frac{1}{\sqrt{2}}|010\rangle + \frac{1}{\sqrt{2}}|000\rangle$);
4. Applicazione della porta CNOT con controllo sul qubit 1 e target sul qubit 2, creando una coppia di qubit entangled ($|\psi\rangle = \frac{1}{\sqrt{2}}|110\rangle + \frac{1}{\sqrt{2}}|000\rangle$);
5. Misura del qubit 1, con memorizzazione del risultato nel registro classico `ro` al bit 0.

Idealmente, quello che si vorrebbe ottenere è una rappresentazione in memoria del programma in grado di contenere diverse istruzioni rappresentanti le varie righe commentate sopra, mutabile attraverso l'aggiunta progressiva di nuovi gate al circuito e traducibile in stringa per l'invio a una QVM o a un Computer Quantistico reale.

Composizione di un programma quantistico

Come accennato in precedenza, Rigetti Computing mette a disposizione una libreria Rust, `quil-rs`[2], che fornisce un'astrazione per il linguaggio Quil pensata per rappresentare programmi quantistici come strutture dati Rust e di manipolarli tramite metodi dedicati.

Il cuore di questa libreria è lo struct `Program`, che rappresenta appunto un programma Quil e che contiene - oltre ad altre informazioni utili - un vettore di `Instruction`:

```rust
pub struct Program {
  // --snip--
  instructions: Vec<Instruction>,
  // --snip--
}
```

Ognuna di queste `Instruction` rappresenta una singola istruzione del linguaggio come variante di enum; a titolo di esempio, le varianti utili a costruire un mapping con il programma riportato sopra sarebbero:

[2]Rigetti & Co LLC, «quil-rs», s.d., https://crates.io/crates/quil-rs.

```
pub enum Instruction {
  // --snip--
  Declaration(Declaration),
  // --snip--
  Gate(Gate),
  // --snip--
  Measurement(Measurement),
  // --snip--
}
```

In linea di principio, sarebbe dunque possibile riprodurre il circuito proposto sopra scrivendo il seguente codice:

```
let mut program = Program::new();
program.add_instruction(
    Instruction::Gate(Gate::new("I", vec![], vec![ Qubit::Fixed(0) ], vec![]).unwrap())
);
program.add_instruction(
    Instruction::Gate(Gate::new("H", vec![], vec![ Qubit::Fixed(1) ], vec![]).unwrap())
);
program.add_instruction(
    Instruction::Gate(
        Gate::new("CNOT", vec![], vec![ Qubit::Fixed(1), Qubit::Fixed(2) ], vec![]).unwrap()
    )
);
```

Dato però che tale approccio non sarebbe affatto ideale per un uso di tutti i giorni, quello che si è deciso di fare al fine di facilitare l'uso da parte degli utenti finali è definire degli struct che vadano a rappresentare le istruzioni più comuni - come l'invocazione dei gate di Pauli e di Hadamard - e implementare per loro il trait `Into<Instruction>` per gestire la conversione nei tipi accettati da quil-rs. A titolo di esempio, ecco l'implementazione del gate H:

```
pub struct H {
  qbit: u64
}

impl H {
  pub fn new(qbit: u64) -> Self {
    Self { qbit }
  }
}

impl Into<Instruction> for H {
  fn into(self) -> Instruction {
    Instruction::Gate(
      Gate::new("H", vec![], vec![ Qubit::Fixed(self.qbit) ], vec![]).unwrap()
    )
  }
}
```

In questo modo, ripetendo l'operazione anche per le istruzioni di dichiarazione e misura, il programma può ora essere riprodotto come segue:

```
let mut program = Program::new();
program.add(Declaration::new("ro".into(), 1));
program.add(I::new(0));
program.add(H::new(1));
program.add(CNOT::new(1, 2));
program.add(Measurement::new(1, "ro".into(), 0));
```

Dove `add` non è altro che una funzione in grado di accettare qualunque tipo che implementa `Into<Instruction>` e gestire automaticamente la conversione:

```
fn add<I: Into<Instruction>>(&mut self, component: I) {
  self.add_instruction(component.into());
}
```

Interazione con compilatore e QVM

A questo punto, creato un programma quantistico, è necessario mandarlo in esecuzione.

Per quanto chiamare il metodo `to_quil()` sul programma appena composto ci consentirebbe di ottenere una stringa pronta per essere caricata manualmente sulla QVM, il requisito di effettuare con facilità analisi dei dati di output ci impone di implementare un'interazione più avanzata con il target e, dunque, di esporre all'utente API più avanzate.

Fortunatamente, un po' come per `quil-rs`, Rigetti mette a disposizione la libreria `qcs`[3] al fine di facilitare la comunicazione con le sue implementazioni di compilatore e QVM.

Un po' più «frammentata» rispetto a `quil-rs` (in quanto richiede all'utente di preparare diverse risorse per il collegamento), la libreria `qcs` vede come suo punto focale l'`Executable`, una versione del programma pronta - appunto - da mandare in esecuzione:

```
let client_config = ClientConfigurationBuilder::default().build().unwrap();
let rpqc_client = Client::new(client_config.quilc_url()).unwrap();
let qcs_client = Qcs::with_config(client_config);
let http_client = HttpClient::from(&qcs_client);
Executable::from_quil(program.to_quil().unwrap())
    .with_quilc_client(Some(rpqc_client))
    .execute_on_qvm(&http_client).await.unwrap();
```

Anche qui, però, si è reso necessario un lavoro di semplificazione per consentire all'utente finale di lanciare un programma senza doversi occupare eccessivamente dei dettagli a basso livello.

Prima di tutto, prevedendo anche la necessità futura di creare altri «esecutori» (banalmente, quello per inviare il programma ad un computer reale), si è definita la seguente interfaccia che dovrà essere rispettata:

```
impl QuantumComputer {
    pub async fn run(&mut self, program: Program) -> anyhow::Result<()> { /* ... */}
    pub async fn get_raw_wavefunction(&mut self, program: Program) -> anyhow::Result<Vec<f64>>
{ /* ... */ }
    pub async fn get_wavefunction(&mut self, program: Program) -> anyhow::Result<Wavefunction>
{ /* ... */ }
}
```

Più nel dettaglio, tali funzioni hanno i seguenti compiti:

- `run` si occupa di mandare in esecuzione sul `QuantumComputer` che si è costruito il programma che viene passato come parametro;
- `get_raw_wavefunction` calcola la funzione d'onda prodotta dal cirucito ricevuto, ritornando un vettore di float da interpretare a coppie (ognuna delle quali rappresenta un complesso);
- `get_wavefunction` fa la stessa operazione di `get_raw_wavefunction` ma produce come risultato uno struct che rende più facile l'elaborazione della funzione d'onda in Rust (ma si rivelerebbe pressochè inutile nella FFI).

A questo punto, nascondendo i dettagli implementativi (quali HTTP client e RPQC client) in un builder in grado d'istanzare un tipo di `QuantumComputer` definito ad-hoc (nello specifico, `QCSComputer`), l'utente finale potrà scrivere ed eseguire il suo programma in poche e semplici righe:

```
let mut computer = QuantumComputerBuilder::using_qcs().build().await;
let mut program = Program::new();
program.add(Declaration::new("ro".into(), 1));
program.add(I::new(0));
```

[3] Rigetti & Co LLC, «qcs», s.d., https://crates.io/crates/qcs.

```
program.add(H::new(1));
program.add(CNOT::new(1, 2));
program.add(Measurement::new(1, "ro".into(), 0));
computer.run(program).await.unwrap()
```

Integrazione con Azure Quantum

Come discusso precedentemente, uno degli obiettivi principali di questo progetto riguarda la possibilità di eseguire i programmi quantistici scritti con CrossQuil su hardware reale.

Per raggiungere questo scopo, si è deciso di integrare il framework con Azure Quantum, una piattaforma cloud di Microsoft che offre accesso a diversi computer quantistici e simulatori, supportando Quil e l'hardware Rigetti out-of-the-box (caratteristica che la rende un candidato ottimale per gli scopi del progetto).

L'integrazione con Azure Quantum è stata realizzata attraverso l'uso delle API REST fornite dalla piattaforma, opportunamente incapsulate per fornire all'utente un'esperienza indistinguibile da quella offerta dalla QVM locale.

Funzionamento di Azure Quantum

Azure Quantum, al pari di molti altri servizi cloud di Microsoft, si basa su una forte integrazione con gli Storage Account, utilizzati come backend per la gestione dell'I/O.

Nel nostro caso in particolare, il pattern per l'interazione tra CrossQuil e Azure Quantum includerà i seguenti passaggi:

1. Creazione di un container all'interno dello Storage Account per l'archiviazione dei file di input e output;
2. Upload del file di input (contenente il programma Quil) sullo Storage Account in un blob con un nome univoco;
3. Invio di una richiesta HTTP POST ad Azure Quantum per l'esecuzione del job, specificando il blob di input e il target hardware desiderato;
4. Polling periodico dello stato del job tramite richieste HTTP GET fino al completamento;
5. Download del file di output (contenente i risultati dell'esecuzione) dal blob specificato nella risposta al completamento del job.

Organizzazione del client

Dal punto di vista dell'implementazione vera e propria, l'elemento fondamentale è lo struct `AzureQuantumWorkspace`, al cui interno sono memorizzate tutte le informazioni per l'interazione con l'API di Azure Quantum e che si occupa di curare l'invio delle varie richieste:

```
pub struct AzureQuantumWorkspace {
    http_client: Client,
    subscription_id: Uuid,
    region: String,
    rg_name: String,
    ws_name: String,
    api_key: String
}
```

Seguendo l'ordine dei passaggi descritti sopra, la prima operazione d'interesse al momento dell'esecuzione vera e propria del programma è l'upload del suo codice sorgente:

```
pub async fn upload_program(&self, program: Program) -> anyhow::Result<Uuid> {
    let program_uuid = Uuid::new_v4();
    let sas_uri = self.get_sasuri("crossquilprograms",
Some(&program_uuid.to_string())).await?;

    let res = self.http_client
        .put(&*sas_uri)
        .header("x-ms-version", "2025-07-05")
        .header("x-ms-blob-type", "BlockBlob")
        .body(program.to_quil()?)
        .send().await?;
```

```
    if let StatusCode::CREATED = res.status() {
        Ok(program_uuid)
    } else { /* --snip-- */ }
}
```

L'operazione di upload, come si può osservare dal codice, avviene all'interno del container standard «crossquilprograms», utilizzando un blob identificato da un UUID generato casualmente per evitare possibili conflitti.

È importante sottolineare il ruolo della funzione `get_sasuri`, che si occupa di ottenere un SAS token per il blob selezionato, consentendo così l'upload senza che l'utente debba fornire direttamente le credenziali dello Storage Account.

Una volta eseguito l'upload è quindi possibile procedere alla creazione del job vero e proprio, all'interno del quale si specifica l'indirizzo del blob con il programma di input e del container in cui posizionare il file di output («crossquiloutputs»):

```
pub async fn enqueue_job(&self, program_uuid: &Uuid) -> anyhow::Result<QuantumJob> {
    let job_uuid = Uuid::new_v4();
    let input_sas_uri = self.get_sasuri("crossquilprograms",
Some(&program_uuid.to_string())).await?;
    let outputs_sas_uri = self.get_sasuri("crossquiloutputs", None).await?;
    let output_sas_uri = self.get_sasuri("crossquiloutputs",
Some(&program_uuid.to_string())).await?;

    let res = self.http_client
        .put(format!("{}/jobs/{}", self.get_baseurl(), &job_uuid))
        .header("x-ms-quantum-api-key", &self.api_key)
        .query(&[("api-version", "2024-10-01-preview")])
        .json(&json!({
            // --snip--
        }))
        .send().await?;

    if let StatusCode::OK = res.status() {
        Ok(res.json::<QuantumJob>().await?)
    } else { /* --snip-- */ }
}
```

Creato il job è ora necessario attendere che Azure lo porti a compimento per poi leggere i dati di output una volta pronti. In particolare, il codice seguente traduce la risposta JSON prodotta da Azure Quantum in una istanza di `RegisterMap`, il che garantisce una perfetta compatibilità con l'output prodotto dalla QVM locale:

```
pub async fn fetch_job_output(&self, job: QuantumJob) -> anyhow::Result<RegisterMap> {
    if job.status.has_succeeded() {
        let res = self.http_client
            .get(job.output_data_uri)
            .header("x-ms-version", "2025-07-05")
            .send().await?;

        if let StatusCode::OK = res.status() {
            let register_map = res.json::<HashMap<String, Vec<Vec<i64>>>>().await?
                .into_iter()
                .map(|(key, val)| (key, RegisterMatrix::Integer(array_from_vecvec(val))))
                .collect();
            return Ok(RegisterMap::from_hashmap(register_map));
        } else { /* --snip-- */ }
    }

    Err(anyhow::anyhow!("The job hasn't succeeded (yet?)"))
}
```

Creazione della FFI

Ora che è stata implementata l'astrazione del linguaggio Quil e della QVM oltre che l'integrazione con Azure Quantum, è possibile procedere con il passo fondamentale per il completamento del progetto: la creazione della Foreign Function Interface (FFI) che permetterà di esporre le funzionalità della libreria Rust appena creata a linguaggi di programmazione esterni.

Scelte architetturali

Quando ci si trova nella condizione di scrivere una FFI in Rust, l'approccio suggerito è quello di creare funzioni che seguano la Application Binary Interface (ABI) di C; in questo modo, infatti, si potrà sfruttare la diffusa compatibilità che altri linguaggi di programmazione garantiscono nei suoi confronti.

Nel rispetto del funzionamento del linguaggio C, il quale non supporta nativamente concetti come classi e metodi, la FFI di CrossQuil è stata realizzata esponendo una serie di funzioni che operano su puntatori a strutture allocate in Rust (e che, dunque, appaiono come «black box» al suo esterno); in questo modo risulterà possibile simulare il comportamento di oggetti e metodi, permettendo agli utenti di creare, manipolare ed eliminare istanze delle strutture definite nella libreria Rust.

Si noti tuttavia come uno dei problemi principali nella scrittura di una FFI in Rust sia la gestione della memoria. Dato infatti che Rust si basa su un sistema di ownership e borrowing per garantire la sicurezza della memoria a compile-time, è fondamentale assicurarsi che le risorse allocate vengano liberate correttamente per evitare memory leak o dangling pointer, tanto più che le chiamate a funzioni di `libc` come `free()` non sono in grado di gestire la memoria allocata da Rust e si renderà dunque necessario esporre delle funzioni dedicate per la deallocazione di ogni struttura esposta.

Esposizione di una struttura dati

Partendo dunque dalla struttura dati più importante della libreria, `Program`, procediamo ad analizzare come è possibile procedere con la sua esposizione.

```rust
#[unsafe(no_mangle)]
pub extern "C" fn qprogram_new() -> *mut Program  {
    Program::new().export()
}

#[unsafe(no_mangle)]
pub extern "C" fn qprogram_add(program: *mut Program, component: *mut Instruction) {
    let program = unsafe { &mut *program };
    program.add(Instruction::import(component));
}

#[unsafe(no_mangle)]
pub extern "C" fn qprogram_drop(program: *mut Program) {
    drop(Program::import(program));
}
```

Queste tre funzioni esemplificano perfettamente il lifecycle di una struttura Rust esposta mediante FFI:

- `qprogram_new` crea una nuova istanza di `Program` e la espone come puntatore mutabile. Questa ultima operazione è effettuata attraverso la funzione export che, definita in un trait `Bridge`, non fa altro che spostare l'istanza sullo heap (attraverso la `Box`) e a far sì che l'allocatore di Rust si «dimentichi» della sua esistenza, producendo un puntatore di cui siamo ora interamente responsabili:

  ```rust
  fn export(self) -> *mut Self {
      Box::into_raw(Box::new(self))
  }
  ```

- `qprogram_add` riprede temporaneamente il controllo della struttura sullo heap «traducendo» il pointer ricevuto in reference mutabile e chiamandovi su di esso il metodo `add`; ovviamente, essendo che `qprogram_add` non acquisisce l'ownership della memoria, questa rimarrà «abbandonata» una volta terminata la sua esecuzione;
- `qprogram_drop` esegue l'operazione simmetrica a `qprogram_new`: «re-importato» lo struct in Rust attraverso la funzione `import` che ne acquisisce la ownership,

  ```
  fn import(external: *mut Self) -> Self {
      *(unsafe { Box::from_raw(external) })
  }
  ```

  procede immediatamente a invocare la `drop` in modo che il gestore della memoria di Rust proceda a deallocarla.

Questo processo viene dunque ripetuto per ogni struct necessario all'uso della libreria Rust precedentemente definita, andando a creare tutte le funzioni necessarie ad esportarli, operarvici sopra e re-importarli per la deallocazione.

Tra queste, risultano particolarmente interessanti le funzioni che espongono le operazioni gestite dal `QuantumComputer`, ovvero la `run` e la `get_raw_wavefunction`, di cui si riporta la seconda:

```
#[unsafe(no_mangle)]
pub extern "C" fn quantum_computer_get_wavefunction(runner: *mut QuantumComputer, program:
*mut Program, len: *mut c_ulonglong, size: *mut c_ulonglong) -> *mut c_double {
    let runner = unsafe { &mut *runner };
    let program = unsafe { &mut *program }.clone();

    if let Ok(rt) = tokio::runtime::Runtime::new() {
        match rt.block_on(runner.get_raw_wavefunction(program)) {
            Ok(mut res) => {
                res.shrink_to_fit();
                let res_len = res.len();
                let res_size = res.capacity();
                let res_ptr = res.as_mut_ptr();
                res.leak();

                unsafe {
                    *len = res_len as c_ulonglong;
                    *size = res_size as c_ulonglong;
                }

                res_ptr
            },
            Err(_) => {
                set_last_error(FFIError::QCSConnectionError);
                null_mut()
            }
        }
    } else {
        set_last_error(FFIError::AsyncRuntimeError);
        null_mut()
    }
}
```

Come è possibile notare, ci sono due particolarità che vengono gestite da questo codice:

- La prima riguarda l'output, che consiste in un vettore di float rappresentanti le componenti delle ampiezze complesse della funzione d'onda. L'esportazione di un `Vec` richiede infatti particolare attenzione: oltre alla memoria «raw» contenente l'array vero e proprio, è necessario gestire anche la lunghezza e la capacità del vettore. Per questo motivo, la FFI si aspetta che vengano forniti due puntatori esterni dove inserire questi valori, che saranno poi indispensabili per una corretta deallocazione della struttura;

- La seconda, dovuta al fatto che le operazioni sul `QuantumComputer` sono asincrone all'interno di Rust, è la gestione della creazione di un ambiente di runtime ad-hoc che vada ad eseguire la `Future` bloccando il thread corrente fino a quando un risultato non sarà pronto per essere restituito al chiamante.

Si noti infine come la funzione di libreria standard `set_last_error` venga utilizzata per comunicare mediante la variabile `errno` quale sia stato l'eventuale errore verificatosi nella chiamata a funzione.

Considerazioni di sicurezza

Come è possibile notare dalla lettura dei frammenti di codice proposti in questi paragrafi, la FFI si aspetta che i puntatori ricevuti siano non-null e rimandino effettivamente a un oggetto del tipo atteso.

Per quanto queste aspettative siano ragionevoli, tanto più che la FFI è pensata per essere utilizzata esclusivamente da programmatori C e da sviluppatori di integrazioni con nuovi linguaggi, tale scelta apre uno spazio per un eventuale miglioramento futuro, incentrato a verificare che tutti gli input ricevuti siano effettivamente non-null e del tipo corretto.

Integrazione di esempio: Java (+ MATLAB)

Completata la libreria base in Rust e scritta la FFI con cui interagirci da altri linguaggi, il prossimo passaggio è stato quello di realizzare un binding per Java, che possa fungere da esempio e da base per eventuali integrazioni con altri linguaggi, oltre che consentire l'uso della libreria direttamente da MATLAB (l'idea originale del progetto).

Al fine di rendere il codice quanto più semplice e vicino alle best-practice per Java, si è deciso di utilizzare Java Native Access (JNA)[4], una libreria in grado di costruire in automatico i binding con la FFI C semplicemente fornendole le firme delle varie funzioni.

Per quanto riguarda il pattern con cui si è proceduto a realizzare l'implementazione, ogni struct disponibile nella libreria Rust troverà un suo corrispettivo in Java, il cui scopo è essenzialmente fare da «wrapper» ai puntatori nudi e crudi e fornire metodi semplificati per interagirvi (gestendo in automatico, ad esempio, la deallocazione della memoria).

Lifecycle di una classe CrossQuil in Java

Per discutere il lifecycle di una di queste classi che abbaimo menzionato, prendiamo ad esempio QProgram, che mostra tutte le caratterstiche dell'interazione tra Java e la FFI di CrossQuil:

```
public class QProgram {
    private interface Bindings extends Library {
        Pointer qprogram_new();
        Pointer qprogram_add(Pointer program, Pointer component);
        Pointer qprogram_drop(Pointer program);
    }

    private static final Bindings BINDINGS = Native.loadLibrary("crossquil", Bindings.class);
    private final Pointer program_ptr;

    public QProgram(Pointer program_ptr) {
        this.program_ptr = program_ptr;
    }

    public QProgram() {
        this(BINDINGS.qprogram_new());
    }

    public QProgram add(Instruction instruction) {
        BINDINGS.qprogram_add(this.program_ptr, instruction.get_raw());
        return this;
    }

    // --snip--

    @Override
    protected void finalize() throws Throwable {
        try {
            if (this.program_ptr != null) {
                BINDINGS.qprogram_drop(this.program_ptr);
            }
        } finally {
            super.finalize();
        }
    }
}
```

Dando un'occhiata al suo codice, la prima cosa che si nota è la presenza di una interfaccia `Bindings`, che estende `Library` e che contiene le firme delle funzioni della FFI C che si vogliono utilizzare; queste

[4]JNA Contributors, «Java Native Access», s.d., https://github.com/java-native-access/jna.

verranno infatti caricate in automatico da JNA, che si occuperà di risolvere i nomi e di effettuare il binding con la libreria nativa traducendo autonomamente i tipi di dati.

La seconda cosa che si nota è la presenza di due costruttori: il primo, che accetta un puntatore raw, è pensato per essere utilizzato internamente quando si ha già a disposizione un oggetto Rust (ad esempio, come risultato di una chiamata a funzione); il secondo, invece, è quello che verrà utilizzato dagli utenti finali per creare un nuovo programma quantistico, e si limita a invocare la funzione `qprogram_new` della FFI per ottenere un puntatore a una nuova struttura dati Rust.

La classe espone poi un metodo `add`, che consente di aggiungere una nuova istruzione al programma quantistico limitandosi ad invocare la funzione `qprogram_add` della FFI, passando il puntatore e l'istruzione da aggiungere (anch'essa ovviamente incapsulata in un oggetto Java ad-hoc).

Infine, nel momento in cui per il garbage collector fosse necessario eliminare l'oggetto, la pulizia della memoria verrà effettuata nel metodo `finalize`, che si occuperà d'invocare la funzione di deallocazione della struttura dati estranea a Java, assicurandosi che non si vengano a creare leak di memoria.

Importazione di array e matrici

Un altro aspetto interessante dell'integrazione con Java è la gestione di array e matrici provenienti dalla FFI.

Queste due strutture dati, infatti, per risultare utili in una qualunque elaborazione dati, non possono essere certo trattate come puntatori opachi, ma devono essere elaborate in modo che siano facilmente utilizzabili dagli utenti finali.

Dato che nel capitolo sulla costruzione della FFI abbiamo commentato l'esportazione della funzione d'onda da parte di `get_raw_wavefunction` nel `QuantumComputer`, vediamo ora la sua controparte in Java:

```
public double[] get_raw_wavefunction(QProgram program) throws Exception {
    try {
        LongByReference len_ptr = new LongByReference();
        LongByReference size_ptr = new LongByReference();

        Pointer wavefunction_ptr = BINDINGS.quantum_computer_get_wavefunction(
            this.runner_ptr, program.get_raw(), len_ptr, size_ptr
        );
        double[] wavefunction = wavefunction_ptr.getDoubleArray(0, (int)len_ptr.getValue());

        BINDINGS.quantum_computer_wavefunction_drop(
            wavefunction_ptr, len_ptr.getValue(), size_ptr.getValue()
        );

        return wavefunction;
    } catch (LastErrorException e) {
        throw new CrossQuilException(e);
    }
}
```

Questo metodo, collocato all'interno della classe `QuantumComputer`, si occupa d'invocare la funzione `quantum_computer_get_wavefunction` della FFI, passando il puntatore al programma quantistico assieme a due riferimenti a variabili di tipo `long` in cui la FFI andrà a scrivere - come descritto in precedenza - rispettivamente la lunghezza e la capacità dell'array che conterrà la funzione d'onda.

Una volta ottenuto il puntatore all'array di `double`, questo può essere facilmente convertito in un array nativo di Java invocando il metodo `getDoubleArray`, che si occuperà di copiare i dati dalla memoria nativa a quella gestita dalla JVM, rendendo poi possibile deallocare immediatamente la prima delle due invocando la funzione `quantum_computer_wavefunction_drop` della FFI.

Passaggio a MATLAB

Come è noto, anche MATLAB viene eseguito direttamente sulla JVM, rendendo possibile l'uso di librerie Java all'interno di script e funzioni MATLAB.

Grazie a questa caratteristica, dunque, sarà possibile utilizzare direttamente le classi Java che abbiamo appena descritto all'interno di MATLAB, rendendoci finalmente in grado di sviluppare e simulare circuiti quantistici direttamente al suo interno.

Sempre a titolo di esempio, ecco come si potrebbe replicare in MATLAB il programma che avevamo commentato nel primo capitolo:

```
program = QProgram() ...
    + I(0) ...
    + H(1) ...
    + CNOT(1, 2);

quantum_computer = QuantumComputerBuilder.using_crossqvm().set_qubits(3).build();
wavefunction = quantum_computer.get_raw_wavefunction(program);

program = program ...
    + Declaration("ro", 1) ...
    + Measurement(1, "ro", 0);

quantum_computer = QuantumComputerBuilder.using_crossqvm().set_qubits(3).build();
res = quantum_computer.run(program).get_register("ro");
```

La CrossQVM

Conclusa l'implemetazione del core di CrossQuil, della sua FFI e perfino dei binding in ben due linguaggi di programmazione «esterni», non si avrebbe torto a considerare il progetto pronto per l'uso ordinario da parte degli utenti finali.

Tuttavia, rimane ancora un piccolo problema legato alla praticità del suo uso: per poter eseguire un programma quantistico, infatti, è necessario disporre di un backend che ne interpreti le istruzioni e ne gestisca l'esecuzione, introducendo una dipendenza sui Quantum Computing Services (QCS) di Rigetti, su una installazione locale della QVM o su Azure Quantum.

Per ovviare a questo problema si è deciso di implementare una versione «embedded» della QVM che possa essere eseguita direttamente all'interno della libreria Rust senza la necessità di dipendere da servizi esterni, rendendo quindi CrossQuil molto più approcciabile da parte di coloro che preferiscono evitare di gestire altro software al di fuori del progetto a cui stanno lavorando.

L'architettura della CrossQVM

Essendo che la macchina virtuale integrata in CrossQuil - che d'ora in poi chiameremo semplicemente CrossQVM - dovrà essere compatibile con Quil, la sua architettura si baserà per forza di cose sul modello di Quantum Abstract Machine (QAM) descritto in *A Practical Quantum Instruction Set Architecture*.

Più nel dettaglio, la QAM non è altro che una 6-tupla $(|\psi\rangle, C, G, G', P, \kappa)$ dove:

- $|\psi\rangle$ è lo stato degli N_q qubit della macchina, inizializzato a $|00...0\rangle$;
- C è una memoria classica da N_c bit, inizializzata a zero;
- G e G' sono due insiemi di gate supportati dalla macchina, il primo di gate statici e il secondo di gate paramatrici;
- P è una sequenza di istruzioni Quil da eseguire;
- $0 \leq \kappa \leq |P|$ è il program counter che indica la posizione della prossima istruzione da eseguire.

La CrossQVM, dunque, dovrà essere in grado di rappresentare e manipolare ciascuno di questi componenti, oltre che di eseguire le istruzioni previste da Quil. Per fare ciò, si è deciso di definire uno struct `QuantumVM` che incapsula lo stato della macchina e ne espone i metodi necessari per interagirvi:

```
pub struct QuantumVM {
    qubits: u32,
    memory: HashMap<String, RegisterMatrix>,
    qmemory: Array1<Complex64>,
    rng: ThreadRng
}
```

dove:

- `qubits` è il numero di qubit supportato dalla macchina;
- `memory` è la memoria classica, organizzata in registri identificati da un nome;
- `qmemory` è la memoria quantistica (lo stato dei qubits);
- `rng` è lo stato del generatore di numeri casuali utilizzato per le misure.

Rappresentazione dello stato dei qubit

La rappresentazione dello stato dei qubit, ovvero $|\psi\rangle$, è forse la scelta che introduce gli aspetti più interessanti dell'implementazione della CrossQVM.

Ad un primo sguardo, infatti, si potrebbe essere tentati a pensare che lo stato di un sistema quantistico possa essere rappresentato come un array bidimensionale di numeri complessi, dove ogni colonna rappresenta un qubit, ovvero:

$$|\psi\rangle_k = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \Leftrightarrow |\psi\rangle_k = \alpha|0\rangle_k + \beta|1\rangle_k$$

Ad esempio:

$$|\psi\rangle = |01\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Un pro di questa rappresentazione sarebbe sicuramente quello di rendere estremamente semplice l'applicazione di gate ad un singolo qubit, in quanto basterebbe moltiplicare la matrice del gate per la colonna corrispondente al qubit su cui si vuole agire; ad esempio, per applicare il gate X sul bit 1:

$$X|\psi_1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Tuttavia, questa rappresentazione non è adatta a descrivere sistemi quantistici, in quanto non tiene conto di ciò che accade in caso di sovrapposizione degli stati; si pensi ad esempio allo stato di Bell $|\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$, che non può essere riscritto come un prodotto diretto di singoli stati base.

La soluzione migliore per la rappresentazione dello stato di un sistema quantistico a N_q qubit è dunque un array monodimensionale di 2^{N_q} numeri complessi, in grado di rappresentare lo spazio di Hilbert dato dal tensore degli spazi dei singoli qubit:

$$|\psi\rangle = \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ ... \\ \alpha_{2^{N_q}-1} \end{bmatrix} \Leftrightarrow |\psi\rangle = \alpha_0|0...00\rangle + \alpha_1|0...01\rangle + \alpha_2|0...10\rangle + ... + \alpha_{2^{N_q}-1}|1...11\rangle$$

Ad esempio:

$$|\psi\rangle = |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \qquad |\psi\rangle = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Lifting di un gate

Nonostante tale rappresentazione si riveli perfetta per descrivere lo stato di un sistema quantistico, essa rende più complessa l'applicazione dei gate, in quanto non è più possibile agire semplicemente sulla colonna corrispondente al qubit interessato.

Essendo infatti che lo stato del singolo qubit è ora «distribuito» su più ampiezze dello stato complessivo che lo coinvolgono, l'applicazione di un gate su di esso avrà effetto su tutte queste e non è dunque sufficiente rappresentarlo con una matrice $2^k * 2^k$ (con k numero di qubit su cui la matrice opera).

Per ovviare a tale problema si è deciso di implementare l'algoritmo di lifting descritto in *A Practical Quantum Instruction Set Architecture* (trattando come qubit 0 il meno significativo): data una matrice U che opera sul qubit k, la sua versione «lifted» $\tilde{U}$ è data da:

$$\tilde{U} = I_{N_q-1} \otimes ... \otimes U \otimes ... \otimes I_1 \otimes I_0$$

dove $\otimes$ è il prodotto di Kronecker, I_i è la matrice identità che opera sul qubit i e U è collocata in posizione $N_q - k - 1$.

Il risultato è una nuova matrice $2^{N_q} \times 2^{N_q}$ che può essere moltiplicata al vettore che rappresenta lo stato.

A titolo di esempio, s'immagini di voler applicare il gate X sul qubit 1 di un sistema a 2 qubit che parte dallo stato $|00\rangle$:

$$\tilde{X}_1 = X_1 \otimes I_0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$\tilde{X}_1|00\rangle = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = |10\rangle$$

Lo stesso procedimento può essere applicato a gate che operano su più qubit adiacenti tra loro, come ad esempio il CNOT:

$$\mathrm{CNOT}_{1,0}\left(\frac{1}{\sqrt{2}}|010\rangle + \frac{1}{\sqrt{2}}|000\rangle\right) =$$

$$= \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \right) \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}|011\rangle + \frac{1}{\sqrt{2}}|000\rangle$$

L'implementazione Rust di questo algoritmo sarà dunque la seguente:

```rust
fn get_rescaled_operation(
    &self,
    gate_name: &str,
    gate_params: &Vec<Expression>,
    raw_target_qubits: &[u64]
) -> anyhow::Result<Array2<Complex64>> {
    let op_matrix = get_common_gates(gate_name, gate_params)?;

    let target_qubits: Vec<u64> = raw_target_qubits.iter()
        .map(|target_qubit| self.qubits as u64 - 1 - target_qubit)
        .collect();

    let id_matrix = Array2::<Complex64>::eye(2);

    let mut rescaled = Array::ones(Dim((1, 1)));
    (0..self.qubits as u64).for_each(|qubit| {
        if let Some(pos) = target_qubits.iter().position(|target_qubit| qubit ==
*target_qubit) {
            if pos == 0 {
                rescaled = kron(&rescaled, &op_matrix);
            }
        } else {
            rescaled = kron(&rescaled, &id_matrix);
        }
    });

    Ok(rescaled)
}
```

Operazioni su qubit non adiacenti

Per quanto le operazioni su qubit adiacenti siano gestite senza problemi dall'algoritmo di lifting, la situazione si complica qualora si renda necessario applicare un gate su qubit non adiacenti tra loro (ad esempio, un CNOT con control su 2 e target su 0).

In questo caso, infatti, non è possibile applicare direttamente l'algoritmo di lifting, in quanto il gate non opera su qubit contigui e dunque non può essere collocato in un'unica posizione all'interno del prodotto di Kronecker.

La soluzione a questo problema, come descritto in *A Practical Quantum Instruction Set Architecture*, consiste nell'utilizzare delle operazioni di SWAP per portare i qubit interessati ad essere contigui tra loro, applicare il gate desiderato e poi riportare i qubit alla loro posizione originale.

Al fine di ottimizzare queste operazioni di permutazione ed evitare di costruire matrici di dimensione $2^{N_q} \times 2^{N_q}$ per ogni SWAP, si è deciso di implementare un algoritmo che operi direttamente sull'array monodimensionale che rappresenta lo stato dei qubit.

Partendo dal principio che, per effettuare lo swap tra due qubit a e b, è sufficiente scambiare tra loro le ampiezze i cui indici differiscono esattamente per i bit in posizione a e b, l'algoritmo procede come segue:

1. Prima di tutto, si enumerano tutti i possibili stati del sistema quantistico, ovvero tutti gli interi da 0 a $2^{N_q} - 1$;
2. Per ognuno di questi si recupera il valore binario a N_q bit che lo rappresenta e da questo si estrae il valore dei bit in posizione a e b;
3. Se sono diversi, si ottiene l'indice dello stato in cui i due bit sono scambiati tra loro e si procede a scambiare le ampiezze corrispondenti.

In Rust, l'implementazione risulterà essere la seguente:

```
fn swap_qubits(&mut self, lhs: &u64, rhs: &u64) {
    let mut new_qmemory = Array1::<Complex64>::zeros(2_usize.pow(self.qubits));

    for config in 0..2_usize.pow(self.qubits) {
        let bit_lhs = (config >> lhs) & 1;
        let bit_rhs = (config >> rhs) & 1;

        let swapped_config = if bit_lhs == bit_rhs {
            config
        } else {
            let config_cleared = config & !(1 << lhs) & !(1 << rhs);
            config_cleared | (bit_lhs << rhs) | (bit_rhs << lhs)
        };

        new_qmemory.index_axis_mut(Axis(0), config).assign(&self.qmemory.index_axis(Axis(0),
swapped_config));
    }

    self.qmemory = new_qmemory;
}
```

In questo modo, dunque, sarà possibile applicare qualsiasi gate su qualsiasi insieme di qubit, anche non adiacenti tra loro, semplicemente effettuando gli opportuni SWAP prima e dopo l'applicazione del gate stesso (ad esempio, portando tutti i qubits non adiacenti in posizione 0 e 1, eseguendo l'operazione e ripristinandoli alla configurazione originale).

Misurazione di un qubit

Altro algortimo fondamentale nella simulazione quantistica è senza ombra di dubbio quello che implementa la misurazione di un qubit.

Come per l'applicazione dei gate, infatti, la rappresentazione dello stato dei qubit come array monodimensionale di ampiezze nello spazio di Hilbert composto rende più complessa la simulazione della misura, in quanto non è più possibile limitarsi ad applicare la regola di Born per il singolo qubit interessato.

Per ovviare a questo problema si fa ricorso ai proiettori e alla formula di misurazione descritta in *A Practical Quantum Instruction Set Architecture*: siano $M_0^k := |0_k\rangle\langle 0_k|$ e $M_1^k := |1_k\rangle\langle 1_k|$ gli operatori di misura per lo stato del qubit k e $\widetilde{M}_0^k$ e $\widetilde{M}_1^k$ le versioni «lifted»; la probabilità di ottenere il risultato $b \in \{0, 1\}$ misurando il qubit k nello stato $|\psi\rangle$ è data da:

$$P\left(M_b^k\right) = \langle\psi|\widetilde{M}_b^{k\dagger}\,\widetilde{M}_b^k|\psi\rangle$$

Che, per un proiettore, si noti essere uguale a:

$$P\left(M_b^k\right) = \langle\psi|M_b^{k\dagger}\,M_b^k|\psi\rangle = \langle\psi|M_b^k M_b^k|\psi\rangle = \langle\psi|M_b^k|\psi\rangle = |\psi\rangle\langle b|b\rangle\langle\psi| = |\langle b|\psi\rangle|^2$$

ovvero esattamente la regola di Born.

Successivamente, scelto l'operatore di misura da applicare sulla base di una estrazione casuale, è possibile procedere con il collasso della funzione d'onda secondo la formula:

$$|\psi\rangle \leftarrow \frac{M_b^k|\psi\rangle}{\sqrt{P\left(M_b^k\right)}}$$

Tutto questo può essere implementato in Rust come:

```rust
fn measure(&mut self, qubit: u64, target: &MemoryReference) -> anyhow::Result<()> {
    let measure_0 = self.get_rescaled_operation("P0", &[qubit]);
    let proj_psi = measure_0.dot(&self.qmemory);
    let zero_prob = proj_psi.mapv(|val| val.conj()).t().dot(&proj_psi).re;

    let measurement = if self.rng.random_range(0.0..1.0) < zero_prob {
        let unitary = measure_0.dot(
            &(Array2::<Complex64>::eye(2_usize.pow(self.qubits)) / zero_prob.sqrt())
        );
        self.qmemory = unitary.dot(&self.qmemory);
        0
    } else {
        let measure_1 = self.get_rescaled_operation("P1", &[qubit]);
        let unitary = measure_1.dot(
            &(Array2::<Complex64>::eye(2_usize.pow(self.qubits)) / (1.0 - zero_prob).sqrt())
        );
        self.qmemory = unitary.dot(&self.qmemory);
        1
    };

    self.memory
        .get_mut(&target.name).ok_or(anyhow::anyhow!("The target memory does not exist"))?
        .as_integer_mut().unwrap()[[0, target.index as usize]] = measurement;

    Ok(())
}
```

Esecuzione di un programma

Una volta implementati gli algoritmi fondamentali per la manipolazione dello stato dei qubit, non rimane che definire il metodo che si occuperà di eseguire un programma quantistico, ovvero di interpretare ed eseguire le istruzioni che lo compongono.

Similmente a quanto avverrebbe in un computer reale, si scompone dunque il programma Quil in una sequenza di istruzioni elementari e si procede ad eseguirle una ad una, aggiornando lo stato della

macchina di volta in volta, ritornando alla fine la memoria classica in modo che possa essere analizzata:

```
pub fn execute(&mut self, instr: &Instruction) -> anyhow::Result<()> {
    match instr {
        Instruction::Declaration(decl) => Ok(self.declare(&decl.name, &decl.size)),
        Instruction::Gate(gate) => self.gate(gate),
        Instruction::Measurement(meas) => {
            if let Some(target) = meas.qubit.try_as_fixed_ref() {
                self.measure(*target, meas.target.as_ref().ok_or(anyhow::anyhow!("Target
memory not specified in measurement"))?)
            } else {
                Err(anyhow::anyhow!("Measurement on non-fixed qubits is not supported yet"))
            }
        },
        _ => Err(anyhow::anyhow!("Instruction {instr:?} is not supported on the CrossQVM
yet"))
    }
}

pub fn run(&mut self, program: Program) -> anyhow::Result<RegisterMap> {
    program.into_instructions().iter()
        .try_for_each(|instr| self.execute(instr))?;

    Ok(RegisterMap::from_hashmap(self.memory.clone()))
}
```

Conclusione e sviluppi futuri

In questa breve tesi si sono volute gettare le basi per lo sviluppo di una piattaforma di programmazione quantistica realmente cross-linguaggio, basata sull'ecosistema Quil e in grado di interfacciarsi con diversi backend hardware e software. A tal fine, è stata realizzata una libreria Rust completa di una QVM integrata che espone una API in stile C tramite la FFI, consentendo così l'integrazione con linguaggi di programmazione diversi da Rust. Inoltre, è stato sviluppato un binding per Java che dimostra la fattibilità dell'approccio.

Tuttavia, il progetto rappresenta solo un primo passo verso una piattaforma quantistica veramente completa. Attualmente, la CrossQVM supporta un sottoinsieme delle istruzioni Quil, focalizzandosi principalmente sulle operazioni quantistiche di base (gate e misure). Un naturale sviluppo futuro potrebbe essere l'estensione del supporto alle operazioni classiche, come l'aritmetica sui registri, le istruzioni condizionali e il controllo di flusso, che sono fondamentali per la scrittura di algoritmi quantistici ibridi e per sfruttare appieno le potenzialità di Quil.

Un altro ambito di espansione potrebbe riguardare la simulazione del rumore (noise) nei circuiti quantistici. L'aggiunta di modelli di noise e di gate non ideali permetterebbe di avvicinare la simulazione software al comportamento reale degli hardware quantistici, rendendo CrossQuil uno strumento prezioso anche per la ricerca sulla tolleranza agli errori e sull'ottimizzazione degli algoritmi.

Bibliografia

[1] Robert S. Smith and Michael J. Curtis and William J. Zeng, «A Practical Quantum Instruction Set Architecture», 17 febbraio 2017. [Online]. Disponibile su: https://arxiv.org/abs/1608.03355

[2] Rigetti & Co LLC, «quil-rs». [Online]. Disponibile su: https://crates.io/crates/quil-rs

[3] Rigetti & Co LLC, «qcs». [Online]. Disponibile su: https://crates.io/crates/qcs

[4] JNA Contributors, «Java Native Access». [Online]. Disponibile su: https://github.com/java-native-access/jna